# Why does Java have primitive data types?

Some object-oriented languages, don't support any primitive data types at all, meaning everything is an object.

But most of the more popular object oriented languages of the day, support both primitive types and objects, as does Java.

Primitive types generally represent the way data is stored on an operating system.

Primitives have some advantages over objects, especially as the magnitude, or number of elements increase.

Objects take up additional memory, and may require a bit more processing power.

We know we can create objects, with primitive data types as field types, for example, and we can also return primitive types from methods.

# Why don't all of Java's collection types support primitives?

But when we look at classes like the ArrayList, or the LinkedList, which we've reviewed in a lot of detail in this section, these classes don't support primitive data types, as the collection type.

In other words **we can't do the following**, creating a LinkedList, using the int primitive type.

This code won't compile.

```
LinkedList<int> myIntegers = new LinkedList<>();
```

This means, we can't use all the great functions Lists provide, with primitive values.

# Why don't all of Java's collection types support primitives?

```
LinkedList<int> myIntegers = new LinkedList<>();
```

More importantly, we can't easily use primitives, in some of the features we'll be learning about in the future, like generics.

But Java, as we know, gives us wrapper classes for each primitive type.

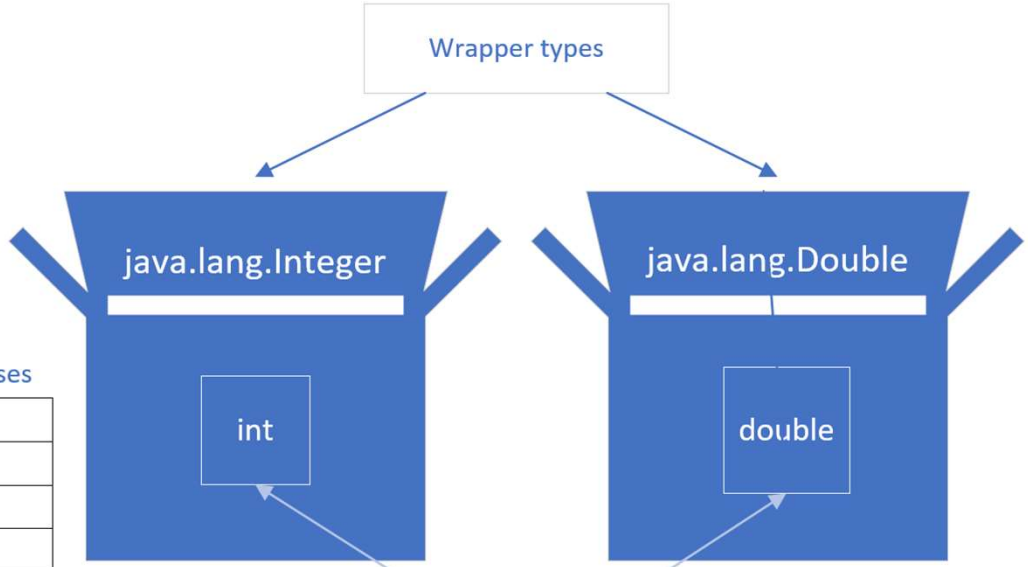
And we can go from a primitive to a wrapper, which is called boxing, or a wrapper to a primitive, which is called unboxing, with relative ease in Java.

# What is Boxing?

A primitive is boxed, or wrapped, in a containing class, whose main data is the primitive value.

Each primitive data type has a wrapper class, as shown on the list, which we've seen before.

Each wrapper type boxes a specific primitive value.




Wrapper Classes

| |
|---|
| Boolean |
| Byte |
| Character |
| Double |
| Float |
| Integer |
| Long |
| Short |

# How do we box?

Each wrapper has a static overloaded factory method, valueOf(), which takes a primitive, and returns an instance of the wrapper class.

The code shown on this slide, returns an instance of the java.lang.Integer class, to the boxedInt variable, with the value 15 in it.

We can say this code **manually boxes** a primitive integer.

```
Integer boxedInt = Integer.valueOf(15);
```

# Deprecated Boxing using the wrapper constructor

Another manual way of boxing, which you'll see in older code, is by creating a new instance of the wrapper class, using the new keyword, and passing the primitive value to the constructor.

We show an example of this here.

```
Integer boxedInt =  new Integer(15);
```

If you try this in IntelliJ, with any Java version greater than JDK-9, IntelliJ will tell you, this is deprecated code.

# Deprecated Code

**Deprecated code** means it's older, and it may not be supported in a future version.

In other words, you should start looking for an alternate way of doing something, if it's been deprecated.

# Using new (with a constructor) is deprecated for wrappers

```
Integer boxedInt =  new Integer(15);
```

Java's own documentation states the following:

- It is rarely appropriate to use this constructor.
- The static factory `valueOf(int)` is generally a better choice, as it is **likely to yield significantly better space and time performance**.

This deprecation applies to all the constructors of the wrapper classes, not just the Integer class.

In truth, we rarely have to manually box primitives, because Java supports something called **autoboxing**.

# What is autoboxing?

We can simply assign a primitive to a wrapper variable, as we show on this slide.

```
Integer boxedInt = 15;
```

Java allows this code, and it's actually preferred, to manually boxing.

Underneath the covers, Java is doing the boxing.  In other words, an instance of Integer is created, and it's value is set to 15.

Allowing Java to autobox, is preferred to any other method, because Java will provide the best mechanism to do it.

# What is autoboxing?

```
Integer boxedInteger = 15;

int unboxedInt = boxedInteger.intValue();
```

Every wrapper class supports a method to return the primitive value it contains.

This is called unboxing.

In the example on this slide, we've autoboxed the integer value 15, to a variable called boxedInteger.

This gives us an object which is an Integer wrapper class, and has the valueof 15.

To unbox this, on an Integer class, we can use the intValue method, which returns the boxed value, the primitive int.

# What is autoboxing?

```
Integer boxedInteger = 15;

int unboxedInt = boxedInteger.intValue();
```

This method is called **manually unboxing**.

And like boxing, it's unnecessary to manually unbox.

# Automatic unboxing

```
Integer boxedInteger = 15;

int unboxedInt = boxedInteger;
```

Automatic unboxing is really just referred to as unboxing in most cases.

We can assign an instance of a wrapper class, directly to a primitive variable.

The code on this slide shows an example.

We're assigning an object instance to a primitive variable, in the second statement.

# Automatic unboxing

```
Integer boxedInteger = 15;

int unboxedInt = boxedInteger;
```

This is allowed, because the object instance is an Integer wrapper, and we're assigning it to an int primitive type variable.

Again, this is the preferred way to unbox a wrapper instance.

Let's get back to some code now, and show different examples of when this feature can be used.